UNITÉ DE RECHERCHE
INRIA-RENNES

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél.:(1) 39 63 55 11

Rapports de Recherche

1992

25ème anniversaire

N° 1692

Programme 1
Architectures parallèles, Bases de données,
Réseaux et Systèmes distribués

A GENERAL SCHEME FOR TOKEN AND TREE BASED DISTRIBUTED MUTUAL EXCLUSION ALGORITHMS

Jean-Michel HELARY
Achour MOSTEFAOUI
Michel RAYNAL

Mai 1992

RR_1692

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE
ET SYSTEMES ALEATOIRES

Campus Universitaire de Beaulieu
35042 - RENNES CEDEX FRANCE
Tél. : 99 84 71 00 - Télex : UNIRISA 950 473 F
Télécopie : 99 38 38 32

Programme 1, projet ADP (Algorithmes Distribués et aPplications)

mai 1992

A general scheme for token and tree based distributed mutual exclusion algorithms

Jean-Michel HELARY, Achour MOSTEFAOUI, Michel RAYNAL

IRISA Campus de Beaulieu - 35042 RENNES cedex - FRANCE

{helary, mostefaoui, raynal}@irisa.fr

Abstract:

In a distributed context, mutual exclusion algorithms can be divided into two families according to their underlying algorithmic principles: those which are permission-based and those which are token-based. Within the latter family a lot of algorithms use a rooted tree structure to move the requests and the unique token. This paper presents a very general information structure (and the associated generic algorithm) for token- and tree-based mutual exclusion algorithms. This general structure does not only cover, as particular cases, several known algorithms but also allows to design new algorithms well suited to topology requirements.

Index terms: distributed algorithm, information structure, mutual exclusion, token, tree structure.

Un schéma général pour les algorithmes d'exclusion mutuelle à jeton fondés sur une arborescence

Résumé :

Dans le cadre des systèmes répartis, les algorithmes d'exclusion mutuelle peuvent être regroupés en deux grandes familles selon leur fondement algorithmique: le principe des permissions ou l'utilisation d'un jeton en unique exemplaire. Un certain nombre d'algorithmes à jeton utilisent une structure arborescente afin de véhiculer les requêtes de demande du jeton. Cet article présente un schéma très général (i.e. une structure d'information et l'algorithme associé) pour les algorithmes à jeton utilisant un telle arborescence. Ce schéma permet non seulement de retrouver et d'expliquer, comme cas particuliers, des algorithmes connus mais également de déduire de nouveaux algorithmes, adaptables à la topologie du réseau sous-jacent. Le schéma proposé constitue donc le modèle générique pour toute une famille d'algorithmes d'exclusion mutuelle à jeton.

Mots-clefs: algorithmique répartie, structure d'information, exclusion mutuelle, jeton, structure arborescente.

1 Introduction

This paper deals with mutual exclusion problem in distributed systems. A distributed system is characterized by a set of nodes, denoted by 1, 2, ... , n. The nodes communicate only by messages exchanged through communication channels; they don't share any memory nor global clock. Channels are supposed to be reliable (messages are neither lost nor corrupted) and communication is asynchronous (message propagation delay is finite but unpredictable). Between any pair of nodes, messages can be delivered out of order (channels can be FIFO or not). Finally, without loss of generality for our purpose, we suppose that there is exactly one process per node: so in the following we consider these two terms as synonyms.

Within such a context, mutual exclusion algorithms can be classified as *permission* or *token* based [8] depending on how the mutual exclusion safety property is achieved (recall that this safety property requires that, at any time, at most one process can be in critical section). The first class is based on the concept of *permission*. A process of identity i (process i for short) which wants to enter the critical section (CS), has to require and obtain permissions from some other processes constituting a set R_i (the requesting set for i). According to the structure of sets R_i, $1 \leq i \leq n$, the class can be divided into two sub-classes. In the first sub-class, conflicts are settled between every pair of nodes and thus permissions are *individual*: when j grants a permission to i, it involves only j. In the second sub-class, permissions have a different meaning and are known as *arbiter* permissions: when j grants a permission to i, this involves all the processes having j in their requesting set; when i will exit the critical section, it will have to restitute this permission to j, in order to let j satisfy other requests. Each of these two kinds of permission entails constraints that must be satisfied by sets R_i [11]. Typical algorithms for these sub-classes are Ricart and Agrawala's [9] for the individual permissions and Maekawa's [4] for arbiter permissions. The liveness property (stating that every request for entering the CS will be satisfied within a finite time) is ensured thanks to a time-stamping mechanism [2] or to the management of an acyclic graph implementing a precedence relation on the processes [1]. These techniques allow to maintain a total order relation on requests, used to prevent cycles in the *wait-for* relation, thus ensuring the satisfaction of all the requests in finite time.

The second class of mutual exclusion algorithms is based on the use of a *token*. Uniqueness of the token guarantees the safety property by subjecting the right to enter the CS to the possession of this token. The main problem which remains to solve is related to the liveness property. Several solutions have been proposed; they differ each from the other in the way the request messages are routed to reach the token. In ring-based algorithms such as [3], the token moves around the ring, granting the right to enter CS to the node currently visited: no explicit request message is needed. In diffusing algorithms [10] requests are sent to all other nodes. In tree-based algorithms, each node sends its requests to one qualified neighbor (its "father") which makes the request progress towards the token [5, 7, 6]. The present paper is devoted to the latter sub-class of distributed mutual exclusion algorithms.

A general scheme for permission-based algorithms has been proposed by B.Sanders [11]; already known particular algorithms such as [9,4] as well as new ones [12] can be deduced from this general scheme. It is based, on the one hand, on the notion of *information structure* (stating which information about the others each process has to maintain), and on the other hand, on the definition of a *general algorithm* using this information structure.

In the present paper, a general scheme (i.e. an information structure and the associated algorithm) is proposed for the class of token-based algorithms using a rooted tree to move the requests. The proposed information structure includes, in particular, a dynamic rooted tree structure logically connecting the nodes involved in the system, and a behavior attribute (*transit* or *proxy*) dynamically assigned to each

node, which is part in the tree evolution. Such a general scheme has not been previously proposed; one of its interesting features is that already known algorithms (cited above), as well as new ones, are contained within its frame: any static or dynamic assignment of the behavior attributes can be considered, each yielding a particular algorithm. Moreover, safety and liveness of the general algorithm are generic properties, in the sense that they imply safety and liveness of all particular algorithms. The rest of the paper contains three parts: the general algorithm is presented in Section 2 and proved in Section 3 (some details are postponed to the Annex 1); particular cases and examples are given in Section 4 (two of them are listed in Annexes 2 and 3). Failures of nodes or channels are not addressed in this paper.

2 The general algorithm

2.1 Principle

Each node is endowed with local variables describing its local state (with regard to the token and to the critical section), its position in the logical rooted tree, and its behavior.

Local state of node *i*

The presence of the token is indicated by the boolean variable $token\_here_i$ whose value is *true* if, and only if, the node *i* has the token. Moreover, the boolean variable $asked_i$ has the value *true* if, and only if, node *i* is currently waiting for the token or executing a critical section. Managing these two variables is rather easy.

Position in the logical rooted tree: asking and returning the token

Each node *i* is endowed with a variable $father_i$ with the following meaning: when a node *i* wants to get the token, it sends a message *request(i)* to its qualified neighbor $father_i$ (then waits for the token arrival). Initially, all *father* variables are set in such a way that they define a rooted tree structure over the nodes, and, as long as there is no request, this structural property remains unchanged. According to the occurrence of requests and to the behavior of nodes, this rooted tree will possibly evolve.

Each node has also a variable $lender_i$; its value indicates the node to which *i* will have to give back the token when leaving the critical section. The two variables $father_i$ and $lender_i$ have "dual" meanings since the former indicates from which node the token should be requested, whereas the latter indicates to which one give back this token (if $lender_i=nil$, the token is kept by *i*). The token message carries either the identity *i* of the node to which it has to be given back (lender) if any, or *nil* if there is no lender; it is denoted by *token(i)* or *token(nil)* respectively.

Let us consider, as an example, a set of nodes logically connected by a star with node *i* as a center, and static values: $\forall j \in \{1..n\} - \{i\}: father_j=i, lender_j=i$. This corresponds to a centralized token-based algorithm: the node *i* receives requests originated by the other nodes and grants them by sending a message *token(i)*; the token is given back to the lender node *i* upon each critical section exit, through the return of a message *token(nil)*.

Behavior of a node

Consider the rooted tree in Figure 1:

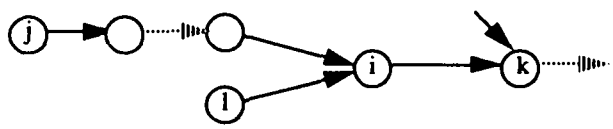

Figure 1. Rooted tree.

The edges of this tree represent the relation ($i, father_i$). Consider the case where a node i, not possessing the token, receives a message *request(j)*. The node i can react to this message with two different behaviors:

- The *transit* behavior means that node i will only forward the message *request(j)* to $father_i$. Afterwards, i considers that, in the future, it will have to send requests to j: consequently it sets $father_i := j$. Moreover, a *transit* node cannot be lender of the token.

- The other behavior, *proxy*, means that node i considers j as its *mandator* and requests the token (to $father_i$) for itself, thus becoming an asking node ($asked_i := true$). When i will receive the token the mandator's request will be satisfied: i will send the token to j and consider its mandate for j as completed. Moreover when a proxy node receives *token(nil)* it becomes its lender. Concerning the variable $father_i$, a proxy node i sets it to k when it receives *token(j)* from k (from now on k is the node to which i will address its requests); when i receives *token(nil)*, it sets $father_i$ to *nil* as it is now the lender.

Each node is thus endowed with a local variable $behavior_i$, having at any time one of the two values: *transit* or *proxy*. It is important to note that this value can change at any time. The possibility of such modification is a fundamental characteristic of the proposed general algorithm. Let us stress, moreover, that these modifications are taken into account but not triggered by the algorithm. Each node is also endowed with a variable $mandator_i$. The value of this variable is a node identity and is meaningful only when i has requested the token for satisfying a request. When $mandator_i = i$ it means that i wants to enter the critical section: whatever the value of $behavior_i$, i is its own mandator; when $mandator_i = j$, $j \neq i$, this means that i has received a message *request(j)* in the context $behavior_i = proxy$. The variable $mandator_i$ will be reset to *nil* when i will receive the requested token: i will cease its mandate for this request. So, when $mandator_i = nil$, this means that the node i has no current request.

Queues

If several nodes j are such that $father_j = i$, the node i can receive several "simultaneous" requests; also, the process associated with the node i may wish to enter the critical section. In order to deal with this multiplicity of requests, a *waiting-queue* is associated with each node. Its service policy is implicit: the only assumption is *fairness*; hereby is meant that every waiting request will be processed in finite time (see §3.4). For example, the FIFO policy is fair. No waiting request can be processed by i unless the boolean variable $asked_i$ has the value *false*. Thus, each node can be seen as a *request server*, whose busy periods correspond to the time during which $asked_i$ is *true*, service corresponds to the request of the token (on current mandator's account), and clients are pending requests waiting in the queue. In the algorithmic expression, the primitive "**wait (not** $asked_i$)" expresses the precondition to the execution of actions related to events *local call to enter_cs* and *receive request(j)*; it corresponds to the fact that process i is occupied to serve another request.

Example

The set of variables defined above constitutes the *information structure* of token algorithms based on a rooted tree. The following example throws light upon their management. The Figure 2 below depicts the initial situation. Node numbered 8 wishes to enter the critical section, and the token is kept by node numbered 1; only nodes belonging to the oriented path (defined by the successive variables *father*) linking node 8 to node 1 are drawn. Nodes 3, 5 and 6 are supposed to be permanently *proxy* (they are circled in the figures), whereas nodes 1, 2, 4, 7, 8 are supposed to be permanently *transit*.

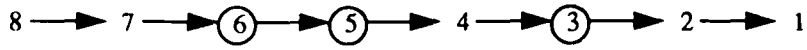

Figure 2. Initial situation.

Node 8 wishes to enter the critical section and **not** *token_here*$_8$ and **not** *asked*$_8$:
send *request*(8) to *father*$_8$=7; *asked*$_8$:=*true*; *mandator*$_8$:=8

Node 7 receives *request*(8) and *behavior*$_7$=*transit* and **not** *token_here*$_7$ and **not** *asked*$_7$:
send *request*(8) to *father*$_7$=6; *father*$_7$:=8

Node 6 receives *request*(8) and *behavior*$_6$=*proxy* and **not** *token_here*$_6$ and **not** *asked*$_6$:
%6 *takes the request on its own account* % send *request*(6) to *father*$_6$=5; *asked*$_6$:=*true*; *mandator*$_6$:=8

Node 5 receives *request*(6) and *behavior*$_5$=*proxy* and **not** *token_here*$_5$ and **not** *asked*$_5$:
%5 *takes the request on its own account* % send *request*(5) to *father*$_5$=4; *asked*$_5$:=*true*; *mandator*$_5$:=6

Node 4 receives *request*(5) and *behavior*$_4$=*transit* and **not** *token_here*$_4$ and **not** *asked*$_4$:
send *request*(5) to *father*$_4$=3; *father*$_4$:=5

Node 3 receives *request*(5) and *behavior*$_3$=*proxy* and **not** *token_here*$_3$ and **not** *asked*$_3$:
%3 *takes the request on its own account* % send *request*(3) to *father*$_3$=2; *asked*$_3$:=*true*; *mandator*$_3$:=5

Node 2 receives *request*(3) and *behavior*$_2$=*transit* and **not** *token_here*$_2$ and **not** *asked*$_2$:
send *request*(3) to *father*$_2$=1; *father*$_2$:=3

Node 1 receives *request*(3) and *behavior*$_1$=*transit* and *token_here*$_1$ and **not** *asked*$_1$:
%1 *gives up the token to* 3 *since its behavior is transit* % send *token*(*nil*) to 3;
father$_1$:=3; *token_here*$_1$:=*false*

Node 3 receives *token*(*nil*) and *mandator*$_3$=5:
% 3 *becomes the lender* % *father*$_3$:=*nil;* send *token*(3) to *mandator*$_3$=5; *mandator*$_3$:=*nil*
% *its mandate for node* 5 *is now completed but asked*$_3$ *remains true* %

Node 5 receives *token*(3) and *mandator*$_5$=6:
father$_5$:=3 % *the token comes from node* 3 %;
% *complete the mandate for node* 6 % send *token*(3) to *mandator*$_5$=6; *mandator*$_5$:=*nil*;
asked$_5$:=*false*

Node 6 receives *token*(3) and *mandator*$_6$=8:
father$_6$:=5 % *the token comes from node* 5 %;
% *complete the mandate for node* 8 % send *token*(3) to *mandator*$_6$=8; *mandator*$_6$:=*nil*;
asked$_6$:=*false*

Node 8 receives *token*(3) and *mandator*$_8$=8:
father$_8$:=6 % *the token comes from node* 6 %;
lender$_8$:=3 % *the token will be returned to node* 3 %; *token_here*$_8$:=*true*
<Critical Section>
send *token*(*nil*) to *lender*$_8$=3; *token_here*$_8$:=*false*; *asked*$_8$:=*false*

Node 3 receives *token*(*nil*) and *mandator*$_3$=*nil*:
token_here$_3$:=*true*; *asked*$_3$:=*false*

At the end of these exchanges, constituting the execution of a single *critical section claim* (CSC

for short) by process 8, the new rooted tree is shown in Figure 3.

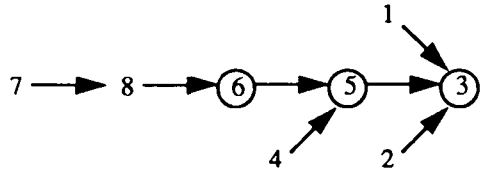

Figure 3. Final situation.

Note that:

- Only the tree concerned with *request* messages is modified; moreover, the modifications depend only on the behavior of nodes. At the extreme two situations can be considered: all the nodes are *proxy*, or all the nodes are *transit*; in the latter case, node 8 will keep the token when exiting the critical section.
- The token, initially kept by the root (node 1 in Figure 2), will be kept by the new root (node 3 in Figure 3) at the end of the CSC.

2.2 The algorithm

The six local variables of each node are all bounded. Two of them are boolean (*token_here*, *asked*), *behavior* has only two values, and the others (*father*, *lender*, *mandator*) have their values in the domain $\{1..\ n\} \cup nil$.

Initialization consists in building a rooted tree by setting the *father* variables; the root r keeps the token, i.e. r is the only node such that $father_r = nil$ and $token\_here_r = true$ and $lender_r = r$. All the other *lender* variables are set to *nil*. Moreover, all the *mandator* variables are set to *nil* and *asked* to *false*.

The text of the algorithm describes the actions performed by each node i upon the occurrence of each of the four possible events: i wishes to enter the critical section (*local call to procedure enter_cs*), i exits the critical section (*local call to the procedure exit_cs*), i receives a *request* message, i receives a *token* message. Apart from the precondition **wait (not** $asked_i$) which may delay the beginning of the actions *enter_cs* and *receive request*, each of these four actions is processed atomically, i.e. without interruption.

```
Upon a call to enter_cs by i
begin
   wait (not asked_i);
   asked_i:=true;                                                          (1)
   if not token_here_i then  mandator_i:=i;
                             send request(i) to father_i;                  (2)
                             wait (token_here_i)   % receipt of token sets lender_i %
   endif
end % enter_cs %

Upon a call to exit_cs by i
begin
   if lender_i≠i  then send token(nil) to lender_i; token_here_i:=false endif;   (3)
   asked_i:=false                                                          (4)
end % exit_cs %

Upon receipt of request(j) by i
begin
   wait (not asked_i);
   case of behavior_i=proxy
         begin  % i becomes proxy for j %
            asked_i:=true;                                               (5)
            if token_here_i
               then  % i temporarily lends the token %
                     send token(i) to j; token_here_i:=false             (6)
               else  % i requires the token %
                     mandator_i:=j;
                     send request(i) to father_i                         (7)
            endif
         end
      behavior_i=transit
         begin
            if token_here_i
               then  % i gives up the token %
                     lender_i:=nil;                                      (8)
                     send token(nil) to j; token_here_i:=false           (9)
               else  % i forwards the request %
                     send request(j) to father_i                         (10)
            endif;
            father_i:=j
         end
   endcase
end % request %
```

As far as receipt of the token by a node i is concerned, three cases are to be considered (at that time, $asked_i$ is *true*):

α) i is the lender and the token is given back to i after a loan (at that time, $mandator_i=nil$).

β) this receipt is an answer to a claim by i to enter the critical section (at that time, $mandator_i=i$); variables $lender_i$ and $father_i$ are updated before entering the critical section.

γ) this receipt is an answer to a request made by i on the account of an other node j (at that time, $mandator_i=j$ with $j\neq i,nil$). This means that i was *proxy* when it received *request(j)*; at the present time, it

can be either *proxy* or *transit*. In both cases, it redefines its position in the tree ($father_i$), sets in the token a value depending on its current *behavior* and on the value brought up by the token, and sends the token to its *mandator*.

```
Upon the receipt of token(j) from k by i
   % j is the token lender; if j=nil the token does not have to be given back%
   % here, asked_i is true, see property 1 below %
begin
   token_here_i:=true;                                                    (11)
   case of mandator_i=nil
         begin % case α: return of the token after loan %
            asked_i:=false                                              (12)
         end % of case α %
      mandator_i=i
         begin  % case β: the claim of i will be satisfied %
                % i updates the position variables %
            if  j=nil   then   % the token has no lender, i becomes the lender%
                               lender_i:=i; father_i:=nil               (13)
                        else   % i will have to give back the token %
                               % it updates the path towards lender %
                               lender_i:=j; father_i:=k                 (14)
            endif;
            mandator_i:=nil
         end% of case β %
      mandator_i≠i,nil
         begin  % case γ: i honors the request of its mandator %
                % meanwhile, its behavior can be changed %
            asked_i:=false;                                             (15)
            case of behavior_i=proxy
                  begin
                     if  j=nil   then   % the token has no lender, i becomes the lender and lends the token %
                                        lender_i:=i; father_i:=nil;     (16)
                                        send token(i) to mandator_i;    (17)
                                        asked_i:=true
                                 else   % j is the lender of the token %
                                        father_i:=k;
                                        send token(j) to mandator_i     (18)
                     endif
                  end
               behavior_i=transit
                  begin
                     if  j=nil   then   % the token has no lender %
                                        lender_i:=nil; father_i:=mandator_i;  (19)
                                        send token(nil) to mandator_i         (20)
                                 else   % j is the lender of the token %
                                        father_i:=k;
                                        send token(j) to mandator_i     (21)
                     endif
                  end
            endcase
            mandator_i:=nil; token_here_i:=false                        (22)
         end% of case γ %
   endcase
end % token %
```

3 Proof of safety and liveness properties

3.1 Safety

Since at any time, a node i cannot be in the critical section unless $token\_here_i$ is *true*, safety follows from the property: there exists at most one node i such that $token\_here_i$ is *true*. Now, this property is easily established. By construction, it holds in the initial state. Afterwards, every sending of the token is possible only by a node x satisfying $token\_here_x$ and this sending triggers $token\_here_x$ to *false*; also, a variable $token\_here_y$ cannot be set to *true* unless y receives the token. As the action associated with this reception is atomic, the required property holds.

3.2 Liveness: preliminary properties

Liveness means that every claim to enter the critical section (CSC) will be satisfied within finite time. Recall, this ensures that neither deadlock nor starvation can occur: the system is deadlocked when no node is in the critical section and all the nodes wishing to enter the critical section will be forever prevented to do so; starvation occurs when a node, wishing to enter the critical section, can be forever unable to do so while other nodes enter and exit.

Three invariant properties are established thereafter.

Property 1

Any node i receiving the token satisfies $asked_i$=*true*.

Proof

When a node i receives the token, this event is the consequence of one of the three possible events:

i. the sending of a message *request(i)* corresponding to a local call to *enter_cs* (line 2); at that time, $mandator_i$=i,

ii. the sending of a message *request(i)* by the *proxy* node i when it received *request(j)* (line 7); at that time, $mandator_i$=j,

iii. the loan of the token by the *proxy* node i answering to a message *request(j)* (lines 6, 17); at that time, $mandator_i$=*nil*.

Whatever the event, it sets $asked_i$ to *true* (lines 1, 5, 17) and this value remains until one of the lines 4, 12 or 15 is processed. But, 12 and 15 are part of the atomic action triggered by the receipt of the token, and 15 is processed only when i exits the critical section, hence after i has received the token (this corresponds to the case $mandator_i$=i). Thus, in any case, $asked_i$ holds when i receives the token. ❑

Property 2

There is at most one node r such that $lender_r$=r. For such a node, we have $father_r$=*nil* and $mandator_r$=*nil*.

Proof

- This property holds in the initial state, by construction; r is the root of the tree.
- No node but the only one such that $token\_here_i$ can perform $lender_i$:=i, and it can only occur during the execution associated with the receipt of the token (lines 13, 16); at that time it sets $father_i$:=*nil*, $mandator_i$:=*nil*, and moreover:

i. i has necessarily received *token(nil)* and, every time a node j sends such a message, it has $lender_j \neq j$ (lines 3, 9, 20)

ii. According to the value of $mandator_i$ upon the receipt of *token(nil)*, two cases are to consider:

a. $mandator_i = i$ (line 13): the node i is granted to enter the critical section and, when it exits, keeps the token (since $lender_i = i$) until the next processing of a request message.

b. $mandator_i \neq i$, *nil* (line 16): the node i sends *token(i)* to $mandator_i = j$ whence the latter will perform, upon receipt of the token, $lender_i := i$, $i \neq j$ ❑

Property 3

(i) $\forall i$: $\neg\, asked_i \wedge token\_here_i \Rightarrow father_i = nil$

conversely,

(ii) $\forall i$: $\neg\, asked_i \wedge father_i = nil \Rightarrow token\_here_i$

Proof

(i) $token\_here_i$ becomes *true* when i receives the token (line 8). At that time, $asked_i = true$ (property 1). When the action associated with this receipt is completed, the assertion $\neg\, asked_i \wedge token\_here_i$ cannot hold unless $mandator_i = nil$ (case α): the case β cannot occur since it leaves $asked_i$ to *true*, and the case γ cannot occur since it resets $token\_here_i$ to *false* (line 22). But the case α corresponds to the return of the token after a loan, whence $lender_i = i$ and, from property 2, $father_i = nil$.

(ii) $asked_i$ becomes false upon one of the following events:

- i exits the critical section (line 4). If, at that time $father_i = nil$, two cases are possible according to the context prevailing when i called *enter_cs*:

c1) $token\_here_i$: in that case, $father_i = nil$ when i called *enter_cs* (from case (i)).

c2) $\neg\, token\_here_i$: in that case, i has requested the token and has performed $mandator_i := i$. This corresponds to the case β, whence, upon the receipt of the token, $lender_i$ has been set to i (line 13).

In both situations, i will keep the token upon the exit of critical section, as long as $asked_i$ remains *false*.

- upon the receipt of the token (lines 12, 15). At that time, $token\_here_i$ is set to *true* (line 11). If the line 12 is processed, it corresponds to the case α: the token is not sent, thus $token\_here_i$ remains *true*. If line 15 is processed, it corresponds to the case γ. But at the end of γ, the assertion $\neg\, asked_i \wedge father_i = nil$ is *false*.

❑

3.3 The deep structure of the algorithm: an abstract tree

For the ease of exposition, let's introduce some terminology. We will say that a critical section claim (CSC) is *created* when a node expresses its wish to enter the critical section; this node is called the *sink*. The CSC is *satisfied* when the *sink* enters the critical section, and is *completed* when the token has been returned to the lender. In-between, the CSC is in *execution*; this execution involves some actions. Executions are sequential: at every time, it is materialized either by a *request* or *token* message, or localized on one, and only one, node (the *current node* for this CSC). More precisely, this execution can be split into two phases: the *token searching phase* (or outward phase) displayed in a routing of *request* messages, constrained by possible waiting on busy nodes; the *token routing phase* (or return phase) displayed in a routing of *token* messages, without waiting on nodes. The set of nodes traversed by the latter phase

comprises exactly those nodes which were traversed by the outward phase and were *proxy* at that time.

Consider now the view of a node, which can be concerned with concurrent CSCs. The status of a node i, with regards to these CSCs, depends on its local variables $asked_i$ and $mandator_i$. As said previously, §2.1, it is convenient to consider a node as a requests server. When a receipt of message *request(j)* (or a local call to *enter_cs*) occurs on node i, this node will process the message (if $asked_i$ is *false*) or will keep it in its waiting-queue (if $asked_i$ is *true*). If $\neg\ asked_i$, the node i is idle (involved in no CSC); if, on the contrary, $asked_i$, the node i is currently serving a CSC, either on the account of $mandator_i=j$ if $j\neq nil$, or i is in the critical section if $mandator_i=nil \wedge token\_here_i$, or i expects the return of the lent token if $mandator_i=nil \wedge \neg token\_here_i$. When a message *request(j)* (or, if $j=i$, a local call to *enter_cs* by i) is waiting in the queue, we will say that the node j is *waiting on* i. This waiting will end as soon as i will begin processing node j's waiting request.

An abstract tree

The following binary relation *a_tree*, defined over the set of nodes, captures the situation of nodes with regard to the CSCs in execution.

Definition

$(i,j) \in a\_tree$ if, and only if, one of the following conditions holds:

(a1) $\neg\ asked_i \wedge father_i=j$.

(a2) $asked_i \wedge$ there is a message *request(i)* in transit towards - or waiting on - node j.

(a3) $asked_i \wedge mandator_j=i \wedge i\neq j$.

(a4) $asked_i \wedge$ there is a message *token(k)*, $k\neq nil$, in transit from j towards i.

(a5) $asked_i \wedge father_i=j \wedge token\_here_i$.

Condition (a1) concerns a node i without pending request: $(i,j) \in a\_tree$ means that the next request from i will be addressed to j. Conditions (a2) to (a5) concern a node with a pending request (an asking node); they correspond to the state and the position of the request currently served by i: (a2) or (a3) holds during the token searching phase, (a4) holds during the token routing phase, and (a5) holds while i is in the critical section.

Proposition 1

At any time, the relation *a_tree* is a rooted tree.

The proof of this important proposition is given in Annex 1. In the rest of the paper, a_tree_i will denote the father of i in the rooted tree *a_tree*.

Corollary 1

The path followed by the requests relative to a CSC is acyclic. The same property holds for the token.

Proof

When a node i sends a *request* message to $father_i$, it satisfies $\neg asked_i$ and thus, from (a1), $a\_tree_i=father_i$. The path followed by the successive requests related to a CSC is thus a path in the tree *a_tree*. For the token, the same reason holds: the token is sent by a node to one of its sons in *a_tree*. ❑

3.4 Liveness proof

Liveness can be proved under the following commonly accepted assumptions:

A1). Transit delay of messages is finite (channels are reliable).

A2). No node can be indefinitely in the critical section.

A3). Each node manages its waiting-queue of requests with a fair policy (this assumption means that, if service times are finite, every waiting request will wait only a finite time).

Under these assumptions, the four following lemmas imply liveness.

Lemma 1

Let j be a node such that $asked_j$ cannot remain indefinitely *true*. Then, every node waiting on j will be served after a finite time.

Proof

Since a node cannot send any new request as long as it has a current request not yet satisfied, the number of nodes waiting on j is bounded by n-1 (where n is the total number of nodes). By the assumption of the lemma, time intervals between two successive ends of services on node j are finite (the end of a service occurs when the variable $asked_j$ becomes *false*). Moreover, by (A3) the service policy is fair. Thus, every node waiting on j will be served after a finite time. ❑

Lemma 2

Let r be the root of *a_tree*. Then $asked_r$ cannot remain indefinitely *true*.

Proof

Suppose $asked_r$=*true*; since, by the assumption of the lemma, a_tree_r=*nil*, none of the conditions (a1) to (a5) is satisfied by pairs (r,x), for any x; thus, in particular, $mandator_r$=*nil*. But, $asked_r \wedge mandator_r$=*nil* can be true only in one of the two cases:

- r is in the critical section. It will exit after a finite time, and then $asked_r$ will be reset to *false*.

- r is the lender of the token. In that case, the token will return after a finite time: in fact, when r lends the token, the number of hops required to get the sink is finite (corollary 1), whence the token reaches the sink in finite time (A1); moreover, the action performed upon the receipt of the token doesn't involve any clause **wait**. Thus, the sink can enter the critical section a finite time after r has sent the token; from (A2), the sink will exit the critical section after a finite time, then returns the token back to the lender; this will take one token hop, of finite time by (A1). ❑

Lemma 3

Let i be a node such that $asked_i$ remains continuously and indefinitely *true*. Then, in a finite time after $asked_i$ becomes *true*, there will be a node i_1 such that $(a\_tree_i = i_1) \wedge asked_{i_1}$ holds indefinitely.

Proof

Suppose $asked_i$ becomes *true* and remains so indefinitely. The node i is neither in the critical section (A2), nor the lender of the token (see proof of lemma 2), whence $mandator_i$=*nil*. In other words, i is busy serving a request: this means that i has previously sent a *request(i)* message. Suppose that this request reaches the root in finite time; from lemma 2, the root cannot remain indefinitely asking, thus the token will be sent after a finite time; but, as shown in lemma 2, the number of nodes traversed by the token to reach node i is finite, and the token never waits on a node; whence, the token will reach node i a finite time after the latter's request, and this contradicts the assumption of the lemma. We have just shown that if node i remains indefinitely asking, then there is a request issued by i which cannot reach the root within a finite time. But, from corollary 1, the number of nodes on the path followed by the request is finite; thus, the only remaining possibility is that the request indefinitely waits on a node i_1 belonging to this path. Such a node necessarily verifies $(a\_tree_i = i_1) \wedge asked_{i_1}$ indefinitely. ❑

Lemma 4

No node i can be such that $asked_i$ remains continuously and indefinitely true.

Proof

By contradiction: suppose there is i such that $asked_i$ remains indefinitely *true*. Recursive application of lemma 3 allows to build a path in *a_tree*, say $i, i_1, i_2, ...$ such that each node belonging to this path remains indefinitely asking. From lemma 2, the root cannot belong to this path, and this is a contradiction with the rooted tree structure. ❑

Theorem (liveness)

Every claim to enter the critical section is satisfied in a finite time.

Proof

The theorem is a direct consequence of lemmas 4 and 1. ❑

4 Some particular algorithms

According to the service policy associated with each node and the definition of rules to manage the *behavior* variables, particular algorithms can be deduced from the previous general algorithm.

4.1 Service policies for waiting-queues

The general algorithm associates with each node i an implicit waiting-queue from which an element (waiting request) can be removed provided that the boolean $asked_i$ has the value *false*. The only assumption about the service policy is fairness (assumption A3 in the liveness proof).

FIFO service policy constitutes a simple way to ensure this assumption. Other implementations are also possible; for example, the so-called *lift-policy* consists in putting systematically at the head of queue, the request corresponding to a local call of *enter_cs* generated by node i itself (this greedy policy has been used in [5, 7]).

4.2 Adding rules for behavior variables

Centralized algorithm

If, for each node i, $behavior_i$ is statically fixed to *proxy*, the underlying tree structure is fixed. The root node r is the allocator of the token: each CSC issued by a node i reaches r via the unique path connecting i with r and the token reaches i via the reverse path; when i leaves the critical section, it returns the token to r (see for example the star network example in §2.1).

Algorithm of Naimi and Trehel

If, for each node i, $behavior_i$ is statically fixed to *transit* we obtain a variant of Naimi and Trehel's algorithm [5]. In this algorithm a node issues requests only on its own account and, by anticipation, declares itself as root (updating its variable $father_i$ to *nil*). Such an anticipation is possible here since the algorithm is designed with the assumption that the *proxy* behavior does not exist; this anticipation couldn't be considered in the general algorithm, as a node issuing a request doesn't know *a priori* if it will or will not have to return the token. The text of Naimi and Trehel's algorithm, as an instance of the general one, is given in the Annex 2. Recall that its complexity, in term of number of messages, is $O(log(n))$ in the mean and $O(n)$ in the worst case.

Algorithms of Van de Snepsheut and of Raymond

If the behavior of each node is *transit* when it has the token and *proxy* otherwise, i.e. $behavior_i$=*transit*$\Leftrightarrow token\_here_i$, we obtain the algorithm proposed by Van de Snepsheut [13] as well as by Raymond [7]. The structure of the tree, initially defined, doesn't change, except the direction of the edges. A CSC issued by a node i follows the unique path connecting i to the root and the token follows the reverse path, changing the direction when it traverses the edge; practically, when a node j receives the token, its behavior changes from *proxy* to *transit* and thus, when j sends the token to a node k (at that time, j is k's father), node j updates $father_j$ to k; afterwards, $behavior_j$ is reset to *proxy*. The text of this algorithm, as an instance of the general one, is given in Annex 3. Recall that its complexity, in term of the number of messages, depends on the structure of the initially defined *unrooted* tree; in the worst case, it is equal to $2d$, where d is the diameter of the tree (longest path in the tree): at most d *request* messages are needed to reach the token, and as much to bring the token to the requesting node. It is possible to initially build a tree with d being $O(\log n)$.

Generality of the proposed algorithm

As far as the general algorithm is not restricted to a particular assignment of the variables *behavior* (*proxy* or *transit*) to the nodes, any static or dynamic assignment can be considered, each yielding a particular algorithm. Actually a particular choice for the behavior of nodes can be controlled according to the supposed evolution of the underlying tree (the efficiency of a tree-based mutual exclusion algorithm indeed depends on this structure). In Naimi and Trehel's, the tree can meet any possible configuration, leading to a worst case message complexity $O(n)$; in Raymond's, the structure is fixed and accordingly the amount of work performed by each node depends on its position in this tree.

More generally, the behavior of each node can be defined dynamically in order to fit the topology of the underlying network. For example (see Figure 4), upon receiving a message *request(j)*, node i can be defined as *transit* if there exists a physical link between j and $father_i$; otherwise, it behaves as a *proxy*. Such a rule to define the behavior of a node allows the token to use shorter paths towards j; communication delays are thus decreased as much as the physical network makes it possible.

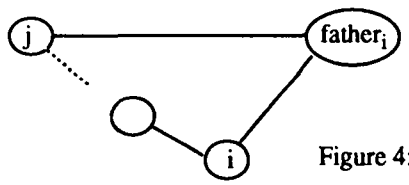

Figure 4: A possible short-cut

Let us still consider another practical situation: suppose we have two networks consisting of nodes $j_1, j_2, \ldots j_a$ and $i_1, i_2, \ldots, i_b$ respectively. Let moreover two nodes of these networks act as gateways. If the gateways are defined as *proxy* each of them only needs to know the identity of the other gateway as far as it considers the other network. Consequently, the algorithm is well suited to composition of networks.

It should be stressed that the assignment of a behavior to a node can be static or dynamic, it can take into account the underlying physical network, the position of the nodes, etc. Thus, the proposed algorithm is very general and can be used to produce particular algorithms better-fitted to a particular situation. It is important to note that whatever are the situation and the criterion chosen to define the behavior of nodes, the resulting algorithm will be correct as a result of the genericity of the proof.

5 Conclusion

In this paper a very general scheme has been presented making a generic model for a class of mutual exclusion algorithms based on the use of a token for safety purpose and on a rooted tree structure carrying the requests for liveness purpose. The interest is twofold: on the one hand, it puts forward the deep structure underlying this class, providing some previously known algorithms (seen as instances of this class) with an explanatory frame; on the other hand, it provides the designer with the possibility to define algorithms better-fitted to particular physical supports.

Acknowledgments

Particular thanks are due to J. Brzezinski for a very careful reading of the manuscript and to M. Mizuno and M. Neilsen who suggested the gateway application. We also thank the French C^3 project devoted to the study of parallelism and distribution for its financial support.

References

[1] K. M. Chandy, J. Misra.
The drinking philosophers problem.
ACM Trans. on Prog. Languages and Systems, Vol. 6,4, (1984), pp 632-646.

[2] L. Lamport.
Time, clocks and the ordering of events in distributed systems.
Comm. of the ACM, Vol. 21,7, (1978), pp 558-564.

[3] G. Le Lann.
Distributed systems: towards a formal approach.
IFIP Congress, Toronto, (1977), pp 155-160.

[4] M. Maekawa.
A $\sqrt{n}$ algorithm for mutual exclusion in decentralized systems.
ACM Trans. on Comp. Systems, Vol. 3,2, (1985), pp 145-159.

[5] M. Naimi, M. Trehel.
An improvement of the log(n) distributed algorithm for mutual exclusion.
Proc. 7th IEEE Int. Conf. on Dist. Comp. Systems, Berlin, (1987), pp 371-375.

[6] M. L. Neilsen, M. Mizuno.
A dag based algorithm for distributed mutual exclusion.
Proc. 11th IEEE Int. Conf. on Dist. Comp. Systems, Austin, (1991), pp 354-360.

[7] K. Raymond.
A tree based algorithm for distributed mutual exclusion.
ACM Trans. on Comp. Systems, Vol. 7,1, (1989), pp 61-77.

[8] M. Raynal.
A simple taxonomy for distributed mutual exclusion algorithms.
ACM Op. Systems Review, Vol. 25,2, (1991), pp 47-50.

[9] G. Ricart, A. K. Agrawala.
An optimal algorithm for mutual exclusion in computer networks.
Comm. of the ACM, Vol. 24,1, (1981), pp 9-17.

[10] G. Ricart, A. K. Agrawala.
Author's response to "On mutual exclusion in computer networks" by Carvalho and Roucairol.
Comm. of the ACM, Vol. 26,2, (1983), pp 147-148.

[11] B. Sanders.
The information structure of distributed mutual exclusion algorithms.
ACM Trans. on Prog. Languages and Systems, Vol. 5,3, (1987), pp. 284-299.

[12] M. Singhal.
A dynamic information structure mutual exclusion algorithm for distributed systems.
IEEE Trans. on Parallel and Distributed Systems, Vol.3,1, (1992), pp.121-125.

[13] J. L. A. Van de Snepsheut.
Fair mutual exclusion on a graph of processes.
Distributed Computing, Vol. 2, (1987), pp 113-115.

Annex 1: proof of the proposition 1.

This proposition states that at any time, the relation *a_tree* is a rooted tree.

1. Initially, the collective variables *father* define a rooted tree; since all the variables *asked* are *false*, relations *a_tree* and *father* are the same.
2. By induction, we show that all the evolutions of the relation *a_tree* maintains the rooted tree structure. Assume that, at a given time, *a_tree* is a rooted tree, and let i and j be two nodes such that $(i,j) \in a\_tree$. Let's examine the five conditions.

1st case. $\neg\, asked_i \wedge father_i = j$.
Two events can disable this condition: $asked_i$ becomes true, or the value of $father_i$ is modified.

The first event necessarily corresponds to the sending of a message *request(i)* to j. In fact, from property 3, $\neg\, asked_i \wedge father_i \neq nil \Rightarrow \neg\, token\_here_i$. After this action, the condition (a2) is satisfied by the pair (i,j) and thus the edge (i,j) remains. The graph *a_tree* is not modified.

The second event necessarily corresponds to the receipt of a message *request(k)* by i; actually thanks to property 1 and to $\neg\, asked_i$ node i cannot receive the token. This event is similar to the one of the next case (with the substitution of (i,j) to (k,i)).

2nd case. $asked_i \wedge$ there is a message *request(i)* in transit towards - or waiting on - node j.

Only one event can disable this relation: the node j begins to process the message *request(i)*. In fact, the variable $asked_i$ cannot change from *true* to *false* as long as this request has not been satisfied. When j begins to process this message we have $\neg asked_j$, and four cases have to be considered according to the state of j:

c1) $behavior_j = proxy$

c11) $token\_here_j$: the node j performs the following actions: (lines 5, 6)
$asked_j$:=**true**; **send** *token(j)* **to** i;
After these actions, condition (a4) holds for the pair (i,j) and thus the edge (i,j) remains. Moreover, none of the (a1) to (a5) condition is satisfied by the pair (j,i) and thus $(j,i) \notin a\_tree$. The graph *a_tree* is not modified.

c12) $\neg\, token\_here_j$: since $\neg\, asked_j$, property 3 ensures $father_j \neq nil$. Relation (a1) is satisfied by the pair $(j\, father_j) \in a\_tree$. Since, by induction, *a_tree* is a rooted tree, we have $father_j \neq i$. The node j performs the following actions (lines 5, 7):
$asked_j$:=**true**; $mandator_j := i$ $(i \neq j)$; **send** *request(j)* **to** $father_j$;
After these actions, relation (a3) is satisfied by the pair (i,j) and relation (a2) by the pair $(j\, father_j)$. The graph *a_tree* is unchanged.

c2) $behavior_j$=*transit*

c21) $token\_here_j$: since $\neg\ asked_j \wedge token\_here_j$, property 3 ensures that $father_j$=*nil* and thus, by the definition of *a_tree*, *j* is the root (Figure 5.1).

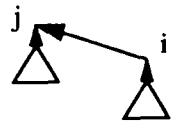

Figure 5.1

Node *j* performs the following actions:
send *token(nil)* **to** *i*; $father_j$:=*i*;
and $asked_j$ remains *false*. After these actions, none of the condition (a1) to (a5) is satisfied by the pair *(i,j)*, but the pair *(j,i)* satisfies (a1). Substitution of edge *(j,i)* to edge *(i,j)* in the rooted tree *a_tree* gives a new tree, whose root is *i* (Figure 5.2).

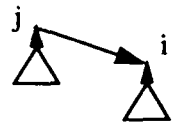

Figure 5.2

c22) $\neg\ token\_here_j$: as for c12, $(j,father_j) \in a\_tree$ and $father_j \neq i$. Let $k=father_j$; the position of the three nodes *i, j, k* in the rooted tree *a_tree* is depicted in the Figure 5.3.

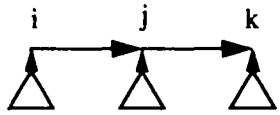

Figure 5.3

Node *j* performs the following actions (line 10)
send *request(i)* **to** $father_j(=k)$; $father_j$:=*i*;
and $asked_j$ remains *false*. After these actions, the condition (a2) is satisfied for the pair *(i,k)*, (a1) for the pair *(j,i)*, and none of the conditions (a1) to (a5) for the pairs *(i,j)* and *(j,k)*. The new graph is depicted in Figure 5.4. and thus *a_tree* remains a rooted tree.

Figure 5.4

3rd case. $asked_i \wedge mandator_j = i \wedge i \neq j$.
Only one event could disable this condition, namely when $mandator_j$ takes a value different from *i* (i.e. becomes *nil*). In fact, $asked_i$ cannot become *false* as long as *j*'s mandate for *i* is not completed. But the modification of $mandator_j$ is bound to the receipt of the token since, as long as it hasn't been received, $asked_j$ remains *true* and thus no new request can be processed by *j*. When *j* receives *token(p)* (from some *k*), its local context satisfies $asked_j$=*true* and $mandator_j \neq j,nil$ thus four cases have to be considered according to the state of *j*.

c1) $behavior_i$=*proxy*

c11) *p=nil:* for any *x*, none of the conditions (a1) to (a5) by pairs *(j,x)* are satisfied, thus *j* is the root of *a_tree*. The node *j* performs the following actions: (lines 16 and 17)
$father_j$:=*nil;* **send** *token(j)* **to** $mandator_j$ *(=i)*

and $asked_j$ remains *true*. After this, j is still root of *a_tree*, and thus the graph of *a_tree* is not changed.

c12) $p \neq nil$: condition (a4) is satisfied by (j,k), hence $(j,k) \in a\_tree$. The node j performs the following actions: (lines 15 and 18)
$asked_j$:=**false**; $father_j$:=k; **send** *token(p)* **to** $mandator_j$ % $mandator_j = i$ %
After these actions, condition (a1) holds for the pair (j,k) and condition (a4) for the pair (i,j).The graph *a_tree* is not changed.

c2) $behavior_i = transit$

c21) $p = nil$: as in c11, j is the root of *a_tree* (Figure 5.5). The node j performs the following actions (lines 15, 19 and 20)
$asked_j$:=**false**; $father_j$:=$mandator_j$ $(=i)$; **send** *token(nil)* **to** $mandator_j$ $(=i)$.

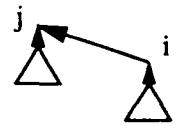

Figure 5.5

After these actions, condition (a1) holds for the pair (j,i) and none of the conditions (a1) to (a5) is satisfied by the pair (i,j). Substitution of edge (i,j) by (j,i) leads to a new tree, with root i (Figure 5.6).

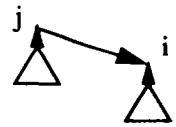

Figure 5.6

c22) $p \neq nil$: the proof is the same as in the case c12 above.

4th case. $asked_i \wedge$ there is a message *token(k)*, $k \neq nil$, in transit from j towards i.
Only the receipt of the token by i can disable this condition: the two terms of the conjunction will become false at the same time. When this event occurs, $asked_i$ is *true* and $mandator_i \neq nil$ since the received message is *token(k)*, $k \neq nil$. There are two cases, according to the value of $mandator_i$:

c1) $mandator_i = i$: node i performs the following actions: (lines 11 and 14)
$token\_here_i$:=**true**; $father_i$:=j;
and $asked_i$ remains *true*. After this, condition (a5) is satisfied by the pair (i,j) and thus *a_tree* doesn't change.

c2) $mandator_i = l \neq i$: condition (a3) is satisfied by the pair (l,i). The configuration of *a_tree* is shown in Figure 5.7.

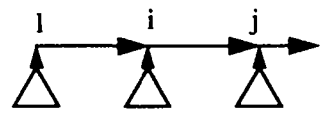

Figure 5.7

Node i performs the following actions: (lines 15, 18 or 21, 22)
$asked_i$:=**false**; $father_i$:=j; **send** *token(k)* **to** $mandator_i$ $(=l)$; $mandator_i$:=*nil*;
After these actions, condition (a4) holds for the pair (l,i), condition (a1) for (i,j) and none of the conditions (a1) to (a5) is satisfied by pairs (i,l) and (j,i). The graph of *a_tree* remains the same.

5th case. $asked_i \wedge father_i = j \wedge token\_here_i$ (the node i is in the critical section).
Only one event can disable this condition: when $asked_i$ becomes *false*. In fact, as long as $asked_i$ is

true, *i* cannot process any request; moreover, having the token, *i* cannot receive it. Thus the terms $father_i=j$ and $token\_here_i$ cannot become *false*. Since $father_i \neq nil$, property 2 ensures $lender_i \neq i$. The node *i* performs the following actions: (lines 4, 3)
$asked_i$:=**false**; **send** *token(nil)* **to** $lender_i$;
After these actions, condition (a1) is satisfied for the pair (i,j) and none of the conditions (a1) to (a5) is satisfied by the pair $(lender_i,i)$. Thus, the graph of *a_tree* remains the same.

□

Annex 2

Below is the text of a variant of Naimi and Trehel's algorithm, deduced from the general algorithm proposed in this paper; all the nodes have a *transit* behavior. Hence, variables $behavior_i$, $lender_i$ and $mandator_i$ are removed, and the *token* doesn't carry any value.

```
Upon a call to enter_cs by i
begin
   wait (not asked_i);
   asked_i:=true;
   if not token_here_i then  send request(i) to father_i;
                             wait (token_here_i);
   endif;
end % enter_cs %


Upon a call to exit_cs by i
begin
   asked_i:=false;
end % exit_cs %


Upon a receipt of request(j) by i
begin
   wait (not asked_i);
   if token_here_i
      then
         send token to j;
         token_here_i:=false;
      else
         send request(j) to father_i;
   endif;
   father_i:=j;
end % request %


Upon a receipt of token by i
begin
   token_here_i:=true;
   father_i:=nil;
end % token %
```

Annex 3

Below is the text of a variant of Raymond's algorithm, deduced from the general algorithm proposed in this paper; all the nodes have a *transit* behavior when they keep the token (*token_here*) and a *proxy* behavior otherwise. Hence, variables $behavior_i$ and $lender_i$ are removed, and the *token* doesn't carry any value.

Upon a call to *enter_cs* **by** *i*
begin
  wait (not $asked_i$**)**;
  $asked_i$:=**true**;
  if not $token\_here_i$ **then** $mandator_i$:=*i*;
    send *request(i)* **to** $father_i$;
    wait ($token\_here_i$);
  endif;
end% *enter_cs* %

Upon a call to *exit_cs* **by** *i*
begin
  $asked_i$:=**false**;
end % *exit_cs* %

Upon a receipt of *request(j)* **by** *i*
begin
  wait (not $asked_i$**)**;
  if $token\_here_i$ % *here equivalent to:* $behavior_i$=*transit* %
    then
      send *token* **to** *j*;
      $token\_here_i$:=**false**;
      $father_i$:=*j*;
    else
      $asked_i$:=**true**;
      $mandator_i$:=*j*;
      send *request(i)* **to** $father_i$;
  endif;
end % *request* %

Upon a receipt of *token* **by** *i*
begin
  $token\_here_i$:=**true**;
  case of $mandator_i$=*i*
      begin
        $father_i$:=**nil**;
      end
    $mandator_i \neq i$
      begin
        $asked_i$:=**false**;
        $father_i$:=$mandator_i$;
        send *token* **to** $mandator_i$;
        $token\_here_i$:=**false**;
      end
  endcase
  $mandator_i$:=**nil**;
end % *token* %

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique

ISSN 0249-6399